VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra telekomunikační techniky

Absolvování individuální odborné praxe

Individual Professional Practice in the Company

2017 Martin Pyščuk

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra telekomunikační techniky

Zadání bakalářské práce

Student: **Martin Pyščuk**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R059 Mobilní technologie

Téma: Absolvování individuální odborné praxe
Individual Professional Practice in the Company

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Student vykoná individuální praxi ve firmě:Projektově.CZ s.r.o.
2. Struktura závěrečné zprávy:
a. Popis odborného zaměření firmy, u které student vykonal odbornou praxi a popis pracovního zařazení studenta
b. Seznam úkolů zadaných studentovi v průběhu odborné praxe s vyjádřením jejich časové náročnosti
c. Zvolený postup řešení zadaných úkolů
d. Teoretické a praktické znalosti a dovednosti získané v průběhu studia uplatněné studentem v průběhu odborné praxe
e. Znalosti či dovednosti scházející studentovi v průběhu odborné praxe
f. Dosažené výsledky v průběhu odborné praxe a její celkové zhodnocení

Seznam doporučené odborné literatury:

Podle pokynů konzultanta, který vedl odbornou praxi studenta

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Zdeňka Chmelíková, Ph.D.**

Konzultant bakalářské práce: Ing. Zdeněk Solnický

Datum zadání: 01.09.2016

Datum odevzdání: 28.04.2017

doc. Ing. Miroslav Vozňák, Ph.D.
vedoucí katedry

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA Fakulta elektrotechniky a informatiky 1D16 OSTRAVA

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 28. dubna 2017 .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26 odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 28. dubna 2017

Projektově.CZ s.r.o.
Kukučínova 10
709 00 Ostrava–Hulváky
IČO: 29460581 DIČ: CZ29460581

. . . . . .

Rád bych na tomto místě poděkoval firmě Projektově.CZ za možnost absolvování odborné praxe. Dále bych rád poděkoval především Ing. Zdeňku Solnickému a Richardu Římanovi, kteří mi po celou dobu odborné praxe předávali své cenné zkušenosti, a poskytli mi odbornou pomoc při zpracování této bakalářské práce.

Abstrakt

Obsahem této bakalářské práce je popis mého působení ve firmě Projektově.CZ formou individuální odborné praxe. Práce je zaměřena na vývoj interaktivních komponent pro výpis úkolů ve službě Projektově.CZ. V bakalářské práci se nachází i další menší úkoly vykonané během praxe. Závěr práce je věnován zhodnocení dosažených výsledků, přínosu daných úloh, uplatněné znalosti získané během mého studia a chybějící znalosti.

Klíčová slova: JavaScript, React, CSS, HTML, Lodash, Immutable, praxe

Abstract

The content of this thesis is a description of my career with the company Projektově.CZ through individual professional practice. The work is focused on the development of interactive components for the listing of tasks in the service Projektově.CZ. The bachelor thesis also includes another smaller tasks carried out during the practice. The end of this thesis is devoted to the evaluation of the achieved results, the benefits of the tasks, the applied knowledge acquired during my studies and the missing knowledge.

Key Words: JavaScript, React, CSS, HTML, Lodash, Immutable, practice

Obsah

Seznam použitých zkratek a symbolů

DOM	–	Document Object Model
JS	–	JavaScript
CSS	–	Cascading Style Sheets
HTML	–	Hyper Text Markup Language
ES	–	ECMAScript
URL	–	Uniform Resource Locator
JSON	–	JavaScript Object Notation
BEM	–	Block Element Modifier
MVC	–	Model View Controller

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

Hlavním důvodem proč jsem se rozhodl vybrat si absolvování odborné praxe, místo výběru jiné bakalářské práce, bylo získání praktických zkušeností. Také mě velmi zajímalo, jaké to je již pracovat jako vývojář ve firmě.

Odbornou praxi jsem vykonal u společnosti Projektově.CZ. Projektově.CZ je menší firma, a díky tomu, bylo možné se úzce podílet na vývoji, vzhledu a na nových funkcích pro službu této firmy. Dalším důvodem výběru této firmy, byla práce s JavaScriptem a frameworkem React, který je v dnešní době velmi používaný.

Úvodní část této bakalářské práce popisuje společnost Projektově.CZ a mojí pozici v této firmě.

V další části bakalářské práce je zadání a nástřel problémů u hlavního úkolu, kterým bylo vymyšlení, a následná implementace interaktivních komponent pro výpis úkolů služby Projektově.CZ.

Následující část bakalářské práce se zaměřuje na popis technologií využitých při vykonávání odborné praxe.

Čtvrtá část je zaměřena na popis implementace a řešení daných problémů u hlavního úkolu.

Další část se zabývá popisem menších úkolů, jejich problematiku a případně jejich řešení.

V závěru uvádím dovednosti získané během studia, které jsem poté využil při vykonávání odborné praxe, dovednosti a znalosti, které jsem díky absolvování této odborné praxe získal, a také zhodnocení, přínos a využití mnou vykonané práce.

2 Představení společnosti a pracovní zařazení

2.1 Základní popis společnosti

Projektově.CZ s.r.o. je ostravská společnost (logo viz Obrázek 1), která dlouhodobě spolupracuje s univerzitou Vysoká škola báňská v Ostravě. Ve spolupráci s odborníky na projektové řízení, ať už z odborných řad, nebo od partnerských firem, poskytuje nástroj pro přehledné řízení firmy, projektů a úkolů.

2.2 Informace o společnosti

Společnost Projektově.CZ vznikla 11.02.2013 na základě technologického transferu ze společnosti Railsformers s.r.o. Hlavní aktivitou společnosti Projektově.CZ je poskytování služby Projektově.CZ – programu pro snadné projektové řízení. Tato služba je cílena na malé a střední firmy, které potřebují řešit efektivní úkolování ve firmě.

Podstatou Projektově.CZ je poskytovat software jako službu, - SAAS (Software As A Service). SAAS je poskytování softwaru přes internet, kdy klient není nucen instalovat program lokálně na počítači. Ke spuštění programu stačí klientovi pouze internetový prohlížeč a přístup na internet. Díky tomu je služba dostupná z práce, z domova, z počítače, tabletu i mobilního telefonu. O provoz, aktualizaci a údržbu systému se stará společnost Projektově.CZ.

Dle analýzy KPMG se procentní příjem těchto služeb v následujících dvou letech zdvojnásobí.

2.3 Popis služby Projektově.CZ

Služba Projektově.CZ řeší otázku efektivního úkolování, které je prováděno pomocí projektového řízení. Služba umožní uživateli získat přehled nad svými pracovními úkoly i nad pracovními úkoly svých kolegů. Služba Projektově.CZ se drží trendu moderních programů, a je proto vytvořen vysoký tlak na jednoduchost užívání a přehlednost rozhraní.

Služba je tvořena pro uživatele, tudíž případné změny a rozšíření jsou komunikovány právě s nimi. Pro snazší přijetí služby, společnost poskytuje pro zaměstnance potenciálních klientů školení projektového řízení a praktického zvládnutí práce v systému. [1]

Obrázek 1: Logo Projektově.CZ

2.4 První dny a mé pracovní zařazení ve společnosti

Na odbornou praxi jsem se hlásil na pozici Vývojář Software As A Service – JavaScript ES6, React. Na tuto pozici jsem musel projít přijímacím řízením.

Na přijímací řízení jsem se s firmou domluvil přes email, kde jsem se také dozvěděl, co si mám na přijímací řízení připravit. Zde jsem si tedy poprvé vyzkoušel pracovat s frameworkem React a připravil jsem si v něm svůj první program.

Na přijímacím řízení jsem přišel s hotovým programem. Byly mi kladeny otázky, jak jsem tento program implementoval, dále jsem tento program rozšířil, a zeptal jsem se na to, co mi nebylo úplně jasné. Po několika dnech jsem se dozvěděl, že jsem byl přijat.

První dny jsem byl ve společnosti seznámen s prostředím, ostatními spolupracovníky a se službou Projektově.CZ. První den jsem připravoval svůj notebook pro práci se službou Projektově.CZ, a dostal jsem zadány své první úkoly.

3 Zadání a nástřel problémů u hlavního úkolu – Interaktivní komponenty pro výpis úkolů

3.1 Zadání

Navrhnout a vytvořit interaktivní komponenty pro výpis úkolů. Tyto komponenty budou umět filtrovat úkoly a seskupovat úkoly do skupin dle zadaných parametrů od uživatele.

Komponenty bude moci uživatel konfigurovat, zvolené nastavení si budou pamatovat, a uživatel bude moci přes URL adresu sdílet své nastavení s ostatními. Zároveň je kladen důraz na vysokou optimalizaci pro nízké využití prostředků jak na serveru, tak v zařízení uživatele.

Řešení musí využívat technologie, které používá služba Projektově.CZ jakými jsou React, Immutable, Lodash a Javascript ES6. Zároveň je nutné, aby spolupracovaly s ostatními komponentami, které již jsou součástí služby Projektově.CZ.

Výsledné řešení je nutné implementovat do služby Projektově.CZ.

3.2 Návrh řešení

První věcí, kterou jsem musel provést, než jsem začal navrhovat řešení, byla analýza starého řešení. Došel jsem k následujícím třem hlavním problémům.

3.2.1 Pomalé vykreslení

Jedním z hlavních problémů starého filtrování a seskupování úkolů byla jeho rychlost.

Ve starém řešení uživatel nakonfiguroval a odeslal formulář se svým filtrem a seskupením. Tato data a všechny úkoly získal server. Na serveru se úkoly vyfiltrovaly, seskupily a vyrenderovala se celá HTML stránka. HTML stránka se poté poslala zpátky na *frontend* a ten ji poté zobrazil. Data nebylo možné vzhledem k jejich dynamice cachovat. Pokaždé, když se data odesílala tam i zpět, jejich velikost nebyla nejmenší, a problémem byla rychlost přenosu dat. U cachování byl problém také s právy. Například pokud je Petr manažerem projektu a Ondra jeho členem, oba mají různý výpis úkolů. Protože manažer vidí více projektů i úkolů než člen týmu, a zároveň má Petr vyšší práva k práci s úkoly. Také úkoly se během dne dynamicky mění, a je potřeba, aby každý uživatel viděl vždy aktuální verzi. Tato data jsou natolik dynamická, že je není možné cachovat.

Momentálně má služba Projektově.CZ více než 1700 uživatelů. Při postupném otevírání dalších zahraničních trhů, se firma velmi pravděpodobně dostane na více než 10000 uživatelů. Díky tomu je dlouhodobě neudržitelné, nechávat takto složitou logiku na serveru. Server má pouze komunikovat přes API rozhraní a poskytovat rychle a bezpečně jednoduchá obecná data, která je možné cachovat. Ve *frontend* části aplikace se pak dle práv pracuje se získanými daty. Trendem pro tvorbu moderních a vysoce interaktivních aplikací je vyšší využívání těchto *frontend* techno-

logií (React, Angular, Knockout), ve kterých je možné i na *frontend* velmi dobře optimalizovat výkon aplikace. Což při renderování na serveru ve stávajícím řešení není možné.

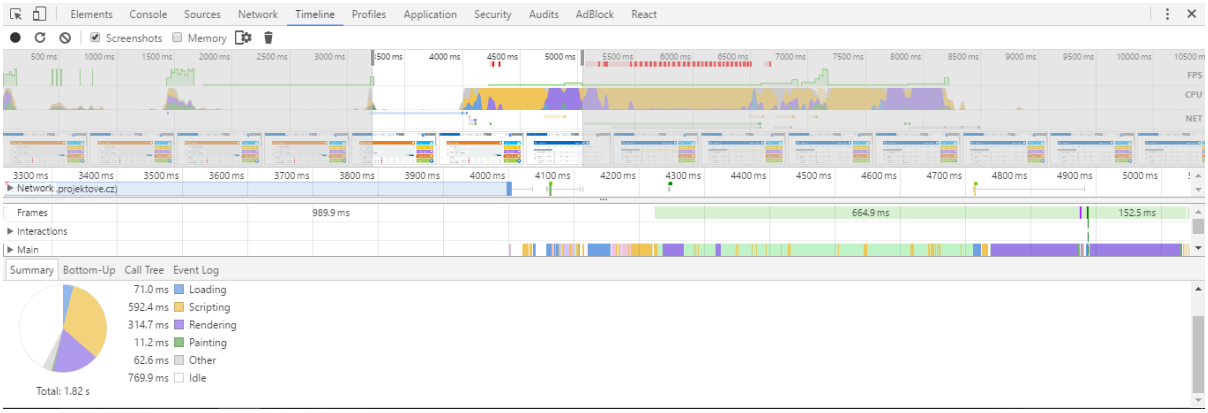
Rychlost požadavku na server při 200 úkolech trval již tak dlouho, že uživatel ztrácel pozornost a navíc se ještě musela vykreslit celá stránka. Zpomalení pro různý počet úkolů jsem změřil, viz Tabulka 1.

U tohoto problému tedy bylo jasné, že se bude muset celá logika, která probíhala na serveru, přesunout na *frontend*.

Tabulka 1: Naměřené časy pro požadavek na server a kompletní vykreslení stránky pro staré filtrování a seskupení úkolů

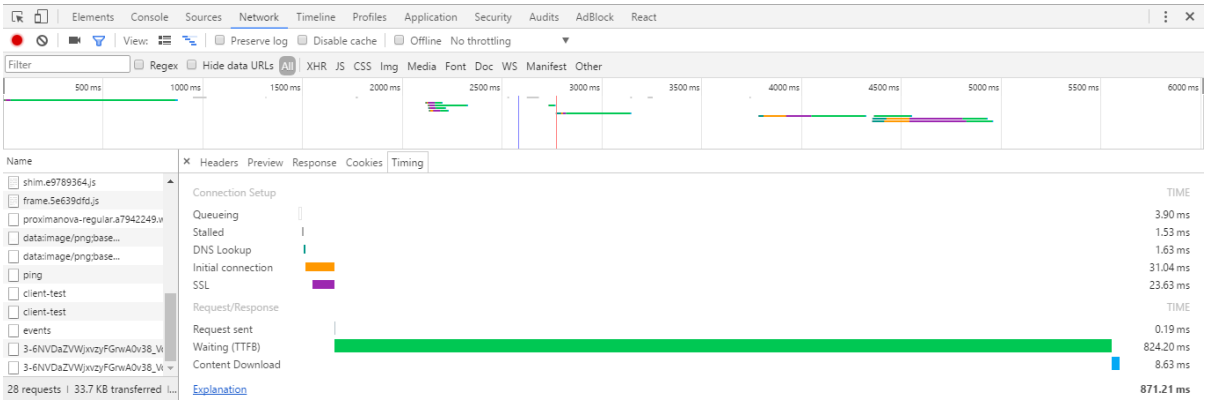
Počet úkolů	**Rychlost požadavku na server** [s]	**Rychlost vykreslení** [s]
25	0,87	1,82
50	1,38	2,45
100	2,60	4,16
200	5,08	7,53

Rychlosti jsem měřil pomocí vývojářských nástrojů prohlížeče Chrome. V záložce *Timeline* jsem měřil celkový čas vykreslení. Odchytával jsem od změny seskupení až po vykreslení stránky, viz Obrázek 2.



Obrázek 2: Měření rychlosti vykreslení pomocí *Timeline*

Další měření jsem prováděl pomocí záložky *Network*, která nám spočítá celkovou dobu trvání požadavku na server, viz Obrázek 3.

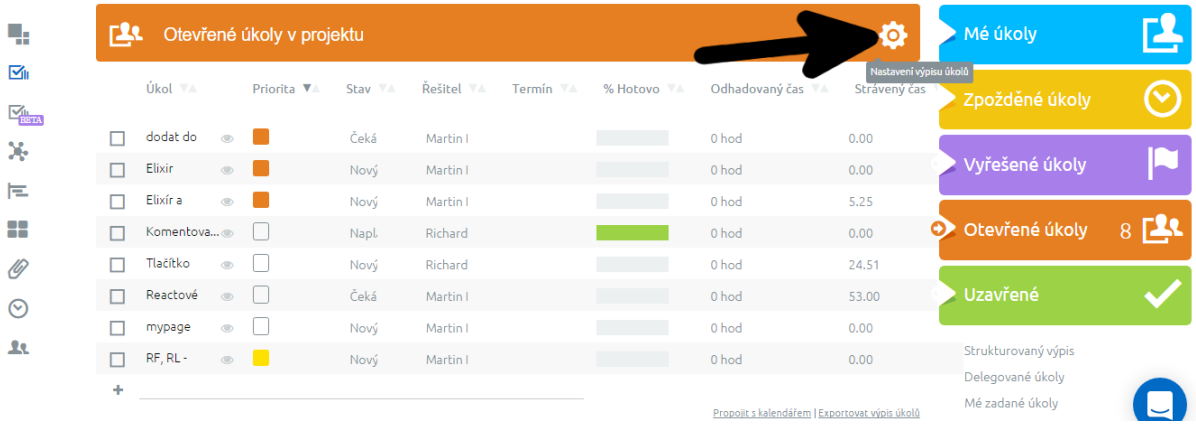


Obrázek 3: Měření rychlosti požadavku na server pomocí *Network*

3.2.2 Špatné uživatelské rozhraní

Jiným velkým problémem bylo staré a špatné uživatelské rozhraní.

Pokud uživatel ve starém řešení chtěl úkoly vyfiltrovat nebo seskupit, musel kliknout na ozubené kolečko u výpisu úkolů, viz Obrázek 4. Toto ozubené kolečko nijak neevokovalo, že díky němu může uživatel dané úkoly vyfiltrovat nebo seskupit.



Obrázek 4: Ozubené kolečko pro otevření rámečku s filtrem a seskupením

Po kliknutí na ozubené kolečko se uživateli zobrazil rámeček, viz Obrázek 5, ve kterém se dalo vybrat seskupení a filtr.

V jeho první části byl *select* na seskupení, kde si mohl uživatel vybrat podle jakého kritéria chce úkoly seskupit. Pokud bylo již seskupení vybrané uživatel, bylo zřejmé pouze podle toho, jak jsou úkoly seskupené. Tedy například Projekt, ale uživatel již přímo neviděl, co daný *select* provádí.

Druhá část už byla viditelně pojmenována názvem Filtry. Bohužel zbytek již nebyl nejlépe vyřešen. Byl zde popisek Přidat filtr a *select*, ve kterém se dal vybrat název filtru. Jenže uživatelé sice viděli typ filtru, ale nikde v tento moment neviděli jeho hodnoty. Ty se zobrazili až po vybrání

typu filtru ze *selectu*. Po vybrání se tedy zobrazil druhý řádek s názvem typu filtru, s logickým operátorem a s hodnotou, kterou chceme zvolit. Logické operátory uživatelé vůbec nevyužívali a byly tak úplně zbytečné. Naopak uživatelům chybělo přidání více řešitelů. Tato možnost ve starém řešení sice byla, ale špatně zpracovaná. Byla řešena pomocí malého tlačítka plus, kterého si nikdo nevšiml. Dalším problémem bylo, že po kliknutí na plus se zobrazil rámeček, ve kterém se dalo vybrat více položek, pomocí podržení klávesy *Control* a vybíráním levým tlačítkem myši. Tohle byla další nevyužitá možnost, jen kvůli špatnému uživatelskému rozhraní a zbytečně složitému ovládání. Další těžkostí bylo, že i když šlo vybrat více položek, fungovaly pouze dvě položky z daného rámečku. I když uživatel vybral více položek.

Poslední chybou tohoto špatně navrženého uživatelského rozhraní bylo, že uživatel, pokud se vyzná a umí přidat filtr, tak po přidání filtru sice vidí vyfiltrovaný výpis. Ale kromě křížku a nápisu Zrušit vlastní řazení (kdy už ani tento nápis neodpovídá k čemu slouží), nevidí jaké filtry a jaké seskupení má nastavené. Aby si uživatel zobrazil, jaké filtry nebo seskupení má nastavené, musí si znovu rozkliknout ozubené kolečko.

Jedním z navrhovaných řešení bylo ponechání rámečku a vylepšení jeho obsahu, ale stále by se tímto nevyřešil problém neviditelnosti, že nějaké filtry a seskupení jsou již nastavené, ani špatný popis ozubeného kolečka. K ozubenému kolečku byla možnost přidat lepší, viditelnější a hlavně jednoznačnější popis, ale vzhledem ke studiím, které dokazují, že lidé nečtou popisky, se tento návrh také neujal.

Obrázek 5: Staré řešení filtru a seskupení

3.2.3 Bez pamatování posledního nastavení

Třetím a zároveň i posledním velkým problémem bylo pamatování si posledního nastavení. Pokud si uživatel pracně nastavil filtr a seskupení, poté odešel z výpisu úkolů na jinou stránku služby Projektově.CZ, a poté se chtěl vrátit na svůj vyfiltrovaný a seskupený výpis úkolů, svůj filtr seskupení najednou už neměl. Výpis sice vypadal, že máte nastavené vlastní filtry a seskupení, ale již nastavené nebyli.

V praxi, majitelé společností reportují úkoly výhradně seskupené dle podprojektů (zakázek) nebo dle řešitelů. Takto při každém otevření projektu nebo zakázky musí filtr znova a zbytečně nastavovat.

Řešením by mohlo být, že by si server pamatoval pouze dané nastavení. Bohužel toto řešení se zavrhlo z důvodu velikosti tohoto nastavení. Také z důvodu, že i kdyby si server dané nastavení pamatoval, stejně by musel vždy filtrovat úkoly znovu a to by bylo příliš pomalé. Jiná možnost zapamatování bohužel nebyla, takže zde byl ještě jeden důvod, proč přesunout celé filtrování a seskupování na *frontend*.

4 Použité technologie

4.1 React

React je řešení pro tvorbu interaktivních uživatelských rozhraní. Efektivně reaguje pouze na změny a překreslují se pouze změněné části.

Výhodou *React* je, že využívá svůj vlastní virtuální DOM. Tudíž všechny změny se propíší nejdříve do něj, a to o poznání rychleji. A až poté se virtuální DOM převede na reálný DOM. [2]

Jinou výhodou *React* je *React Native*, ve kterém se dají psát mobilní aplikace. *React* razí heslo „Naučit jednou, psát kamkoli."

React je založený na komponentách.

4.1.1 Komponenta

Komponenty umožňují rozdělit kód do samostatných, opakovaně použitelných částí. Vytvořit komponentu lze jednoduše pomocí třídy *'React.Component'*, kterou zajišťuje knihovna *React*.

Každá komponenta má životní cyklus. V tomto životním cyklu se volají funkce.

První z nich je *'contructor'*, která se volá ještě před tím, než je komponenta přidána do virtuálního DOMu. Pokud *'constructor'* u komponenty nenapíšeme, používá se defaultní *'constructor'*. Defaultní *'constructor'* je prázdný. Pokud si ale *'constructor'* naimplementujeme sami, máme mnoho možností. Pokud má komponenta rodičovskou komponentu, můžeme ve funkci *'constructor'* zavolat *'constructor'* rodičovské komponenty a díky tomu získat data, která si pošleme z rodičovské komponenty. Tato data jsou v objektu *props*. *Props* by se dle tzv. *React flow* neměl měnit a tato data by měla sloužit pouze pro čtení. Také je možné si vytvořit vlastní data pro komponentu. Tato data jsou v objektu *state*. Poslední, co můžeme ve funkci *'constructor'* udělat je navázat (tzv. *bind*) funkce. Pokud funkci navážeme, uvedeme tím funkci do daného kontextu. Díky tomu je možné ve funkci používat *state* a *props*. Funkce by se neměly zbytečně navazovat.

Další funkcí, která se volá v rámci životního cyklu je *'componentWillMount'*. Tato funkce je volána těsně před přidáním do virtuálního DOMu. Není příliš využívána, místo ní se doporučuje provést operace, které potřebujeme ve funkci *'constructor'*.

Předposlední funkcí životního cyklu je *'render'*. Tato funkce je jediná povinná pro každou komponentu. Funkce *'render'* musí něco vracet. Tato funkce vrací tzv. virtuální DOM, tj. stromovou strukturu, která v sobě nese informaci o tom jak má reálný DOM vypadat. Jednou z možností je jeden *React* element, ve kterém mohou být ostatní elementy. Jinou možnosti jsou *null* a *false*. Tyto možnosti se používají, pokud v nějaké situaci nechceme vracet nic. Funkce *'render'* musí být tzv. *pure* to znamená, že by neměla měnit *state* komponenty. Tato funkce se volá vždy, při změně dat dané komponenty (*state*), při změně dat v *props* nebo při změně dat ve

Store, se kterým daná komponenta komunikuje. Zakázat volání této funkce při změně některých nebo i všech dat je možné ve funkci *'shouldComponentUpdate'*.

Poslední funkcí je *'componentDidMount'*. Tato funkce je volána po přidání komponenty do virtuálního DOMu.

Další funkce, které *React* má se převážně používají pro zaktualizování dat v komponentách.

Poslední funkcí, kterou nám poskytuje *React* komponenta je *'componentWillUnmount'*, která se zavolá poté, co je komponenta odebrána z virtuálního DOMu.[3]

4.1.2 Store a Action

Stores a *Actions* se používají pro komunikaci. Ať už mezi komponentami nebo mezi naší aplikací a serverem.

Store drží většinu dat naší aplikace. Díky tomu jsou data rychleji k dispozici, než kdyby se data stále brali ze serveru. Změnit data ve *Store* lze pouze pomocí *Actions*.

Action poskytují data pro *Store*. Ukládají do něj data, ať už ze serveru nebo z komponenty.

Komunikace tedy probíhá následovně: Komponenta zavolá *Action* a předá ji data, *Action* uloží data do *Store*. Komponenta poté dostane od *Store* nová data. [4][5]

4.2 Immutable

Immutable je knihovna pro *JavaScript* poskytující neměnné datové typy.

Immutable data nelze změnit po vytvoření, což vede k mnohem jednoduššímu vývoji aplikací, je o mnoho rychlejší než *mutable* datové typy. Nemá žádnou obranu proti kopírování a poskytuje snadnější detekci změn.

Poskytuje mnoho struktur, jako jsou: *List*, *Stack*, *Map*, *OrderedMap*, *Set*, *OrderedSet* a *Record*. Tyto datové struktury jsou vysoce efektivní.

Immutable také poskytuje tzv. *lazy* vykonávání, díky čemuž je možné operace nad strukturami řetězit.[6]

4.3 JavaScript ES6

ES6 je standard pro *JavaScript*, který byl vydán v roce 2015. Zde jsou jeho největší novinky.

Arrow funkce, díky nimž lze napsat zkráceně funkci nebo funkci do funkce a podobně. Přibyly také třídy, víceřádkové řetězce, defaultní hodnoty argumentů funkcí. Dále *promise*, který se používá pro odložení operací, než se provede operace před ní (například než dojdou data ze serveru). Také interpolace, což je lepší skládání řetězců, moduly, těmi může být například funkce nebo knihovna, a pomocí export modulu se k němu dostaneme odkudkoli, kde si jej naimportujeme. Nový typ proměnné *let*, jejíž životnost je v rámci daného bloku kódu. Poslední důležitou funkcí je destrukturalizace struktur. [7]

4.4 Lodash

Moderní knihovna pro *JavaScript*, která usnadňuje práci s objekty, poli a podobnými strukturami. Například iterace, manipulace, testování hodnot a vytváření složených funkcí.[8]

5 Vlastní řešení hlavního úkolu – Interaktivní komponenty pro výpis úkolů

V této části popíši, jak jsem řešil dané problémy, a poté podrobný postup řešení.

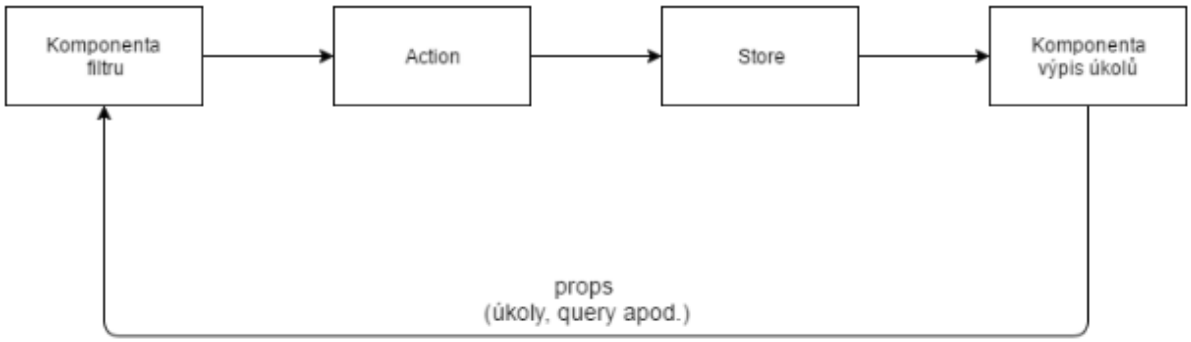
5.1 Řešení problémů

Zde rozeberu dané problémy, popíši, jak jsem je vyřešil. Na obrázcích ukážu, jak vypadá momentální řešení.

5.1.1 Pomalé vykreslení

Problém pomalého vykreslení jsem vyřešil tím, že jsem přestal úplně používat server. Celou logiku vykreslování seskupení a filtrování jsem přesunul na *frontend*, pomocí *React* komponenty, *Action* a *Store*.

Po zadání filtru a seskupení, filtr upravím tak, aby měl danou strukturu. Pošlu jej do *Action*, kde se následně úkoly ve *Store* vyfiltrují a úkoly se pošlou do jiné komponenty, která se stará o výpis úkolů. Díky tomu jsem také dodržel tzv. *React flow*, tedy že komponenta komunikuje se *Store* přes *Action* a *Store* dále posílá data ostatním komponentám pro vykreslení dat, viz Obrázek 6. Co je zde také důležité tak, pokud rodičovská komponenta bere data ze *Store* tak, aby i její potomci nebrali zbytečně data ze *Store*, posílám si data přes *props*. Seskupení bere již daný hotový výpis a seskupí jej do skupin a vrátí komponentě pro výpis úkolů.



Obrázek 6: *React flow*

Kdybych toto řešení popsal stejně jako to staré, tak by vypadalo následovně: Petrovi i Ondrovi jsou zaslána data (úkoly), která vidí, tedy pouze rychlý JSON. Dle jejich práv, viditelnosti, filtru a seskupení jsou již na *frontend* tvořeny pohledy a není nutné neustále komunikovat se serverem.

Porovnání rychlostí starého filtru a seskupování s novým řešením, viz Tabulka 2.

Tabulka 2: Naměřený čas rychlosti vykreslení stránky pro nové řešení, absolutní zlepšení rychlosti a procentuální zlepšení o proti starému řešení

Počet úkolů	**Rychlost vykreslení** [s]	**Zlepšení rychlosti**	
		absolutní [s]	**procentuální** [%]
25	0,40	1,42	54
50	0,66	1,79	73
100	0,91	3,25	78,2
200	1,57	5,96	78,75

U tabulky tentokrát neuvádím rychlost požadavku na server, protože nové řešení již nepracuje se serverem (při seskupení a filtrování). Můžeme si všimnout, že procentuální zlepšení stoupá spolu s počtem úkolů, což je určitě potřeba, jelikož některé firmy pracují s tisíci úkoly.

5.1.2 Lepší uživatelské rozhraní

Pro lepší uživatelské rozhraní jsem musel filtr i seskupení vytáhnout ze starého rámečku, který se zobrazil po kliknutí na ozubené kolečko a umístil jsem jej rovnou nad výpis úkolů. Takto jsou uživateli stále na očích a hned pochopí, k čemu slouží.

U seskupení je i nadpis Seskupit, aby i po nastavení bylo jasné, k čemu daný *select* slouží.

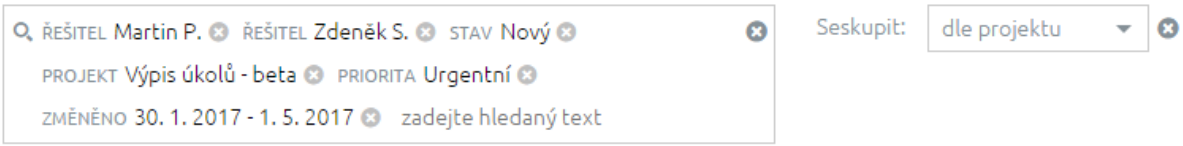
Po přidání filtru se filtr zobrazuje ve formě štítků. Kde nejdříve je napsán typ filtru (řešitel, projekt a podobně) a poté hodnota, takže například jméno uživatele. Smazat filtr jde dvěma způsoby.

První z nich je smazání pouze jednoho nastaveného filtru, a to kliknutím na křížek u štítku.

Druhou možností je větší křížek na kraji rámečku pro štítky, který smaže všechny nastavené filtry.

V rámečku se dá psát i text, který vyhledává v názvu úkolů nebo tak využít našeptávač pro filtry.

V konečném výsledku, tedy uživatel hned nad výpisem úkolů vidí jaké má nastavené filtry a seskupení, viz Obrázek 7.



Obrázek 7: Zobrazení nastaveného filtru a seskupení

5.1.3 Pamatování posledního nastavení

Poslední větší problém bylo pamatování posledního nastavení, ve kterém jsem také řešil, aby se dané nastavení filtru a seskupení dalo poslat kolegovi.

Pamatování jsem řešil pomocí *Local Storage*. V komponentě filtru si ukládám filtr a ten jako strukturu, která má typ a hodnotu (id), ukládám do Local Storage. Ukládání funguje zvlášť pro výpis všech úkolů a pro výpisy úkolů v projektech, viz Obrázek 8.

Obrázek 8: Ukázka *Local Storage* s filtrem a seskupením

Aby bylo možné poslat filtry a seskupení kolegovi, ukládání do *Local Storage* nestačí. Prvním důvodem je, že obyčejný uživatel těžko bude vědět, kde data z *Local Storage* hledat a druhým důvodem je, že data v *Local Storage* máme zakódována, aby nebyla pro uživatele viditelná, ale viditelná pouze pro nás vývojáře, jelikož jsou tato data interní. Proto byla potřeba použít něco jiného než *Local Storage*. Nejlepším řešením mi přišla URL. Jelikož každý uživatel umí zkopírovat a poslat adresu URL někomu jinému. Samozřejmě jsem musel vyřešit, jak mají která data vypadat, a hlavně zkrátit jejich názvy a podobně.

Pro filtr jsem použil písmeno **f**, za kterým je rovná se a jeho filtry ve tvaru **a84**, kde písmeno **a** je typ filtru, zde konkrétně řešitel a **84** je id řešitele. Pro seskupení jsem použil písmeno **g** a za rovnítkem je opět typ, tedy například písmena **pr** pro seskupení dle projektů, viz Obrázek 9.

Obrázek 9: Ukázka URL s filtrem a seskupením

Problém, na který jsem narazil, pokud mi kolega poslal nějaký filtr a seskupení a já již měl nějaký nastavený, tak čemu dát přednost. Nakonec jsem se rozhodl pro URL. Pokud totiž uživatel přejde na stránku s výpisem úkolů standardně bez napsání dalších znaků do URL, tak v URL žádné další informace nejsou. Tyto informace tam vkládám až těsně před načtením stránky ručně. Takže nejdříve zjišťuji, zda se v URL nachází daný *hash* pro filtr a pro seskupení, a pokud ano, tak jsem jej načetl a uložil do *Local Storage*. Pokud v URL nic není, pokusil jsem se načíst filtr a seskupení z *Local Storage*, samozřejmě podle toho, kde se nacházím. Zda v projektu nebo jen ve hromadném výpise úkolů. Vyfiltroval jsem data, seskupil je a zobrazil filtry a seskupení. Pokud nebyl žádný filtr ani seskupení v *Local Storage*, filtr i seskupení byly prázdné, a úkoly se nefiltrovaly ani neseskupovaly.

5.2 Podrobnější postup řešení

Začal jsem tím, že jsem si vytvořil dané komponenty. Ty jsem zahrnul do komponenty obsahující výpis úkolů tak, aby byly správně umístěné na obrazovce.

5.2.1 Seskupení úkolů

Prvně jsem začal pracovat na seskupení, a to hlavně z důvodu, že je tato funkce silným nástrojem pro reporting úkolů. Bylo tedy potřeba ji, co nejdříve dodat. V *selectu* jsou dané výběry seskupení seřazeny dle využití a informací od uživatelů.

V první fázi jsem seskupení vložil pouze jako funkce do komponenty pro výpis úkolů. Problémem, ale bylo, že to zaprvé není správné řešení v rámci *framework React* – rozšiřovat zobrazující komponentu o takto složité funkčnosti. Za druhé rozšíření již tak složité komponenty, ji dělá nepřehlednou. Proto byla potřeba seskupení přepracovat.

Proto jsem navrhl, že seskupení budou jednotlivé komponenty, tedy například komponenta pro seskupení dle projektů, pro seskupení dle řešitelů a podobně. Tato verze byla mnohem lepší a vydržela docela dlouho. Ale problémem byl opakující se kód u každé z komponent. Všechny komponenty měly téměř stejný kód, akorát s jinými daty a s pár změnami. Proto bylo potřeba udělat z nich jen jednu komponentu, která bude velmi dynamická, tak aby zvládala seskupení dle různých dat, které nemají totožnou strukturu.

Posledním řešením (tedy zatím) je, že se ve výpise úkolů vykreslí proměnná *content*, která závisí na tom, zda je nastavená v *selectu* pro seskupení nějaká hodnota. Pokud ne, zavolá se funkce pro obyčejný výpis úkolů. Pokud ano, zavolá se funkce pro seskupený výpis úkolů.

Ve funkci pro seskupený výpis úkolů používám *state* dané komponenty. Pro jeho použití jsem využil novinky v *JavaScript* ES6 a to destrukturalizace, kdy jsem si vytvořil ze *state* konstanty, abych nemusel pokaždé psát například *this.state.issues* pro úkoly, ale aby stačilo napsat *issues*. Druhá využitá technologie v této funkci je *Lodash*. Využil jsem jeho funkcí, které jsem zřetězil. První funkcí byla funkce *'compact'* pomocí, které si vytvořím pole, kde všechny *null* nebo *false* položky jsou odstraněny. Využívám zde také další novinku z ES6 a tím je interpolace. Použil jsem ji, abych složil správný název pro konstantu *Grouped*, a k tomu ještě využívám následující *Lodash* funkci *'upperFirst'* pomocí, které si zvětším první písmeno v řetězci. Do *Grouped* si rovnou posílám data.

Jak již vyplývá z textu, z komponenty se stala konstanta, která je strukturou, jejíž název je *Grouped* a dále v sobě obsahuje daná seskupení. Takže nakonec vypadá struktura například *Grouped[byAuthor]*. Tato konstanta tedy funguje tak, že ji naplním jen danou část její struktury, dle toho, které seskupení je vybráno a to právě, pomocí funkce *'compact'*.

Konkrétní řešení jak naplnit konstantu, řeším v jiném souboru s názvem *'groups'*, aby tato logika zbytečně nebyla v komponentě pro výpis úkolů. Jelikož úkoly jsou data typu *Immutable*, tak v souboru *'groups'* nepoužívám *Lodash* funkce. První funkci, kterou využiji je *'groupBy'* pomocí, které se mi rozdělí úlohy dle daného seskupení – takže například dle jmen autorů, dle termínu a podobně. U jednodušších struktur, jako jsou například priority nebo stavy, používám dále jen jednoduchou funkci *'sort'*, ve které využijí funkce z naší interní knihovny funkcí. A to *'comparator'* pomocí, které porovnávám dané řetězce, čísla a podobně, a vracím zpátky číslo nula pokud jsou stejně velké, jedna pokud je první hodnota větší než druhá a mínus jedna

pokud je první hodnota menší. U složitějších struktur, tedy u všech ostatních, používám místo funkce *'sort'* funkci *'sortBy'*, protože potřebuji porovnat, zda daná hodnota není prázdná, tedy neexistuje. Pokud první hodnota neexistuje, vrátím ji jako větší, aby byla seřazena nahoře. U autorů a řešitelů ještě navíc musím vyřešit problém, kdy existuje tzv. admin uživatel. Pokud je, tak mu nastavím jméno ze slovníku, který firma využívá. Jinak použiji jméno uživatele.

Pokud je seskupení nastavené dle řešitele, ukázka naplnění konstanty *Grouped* je ve Výpisu 1.

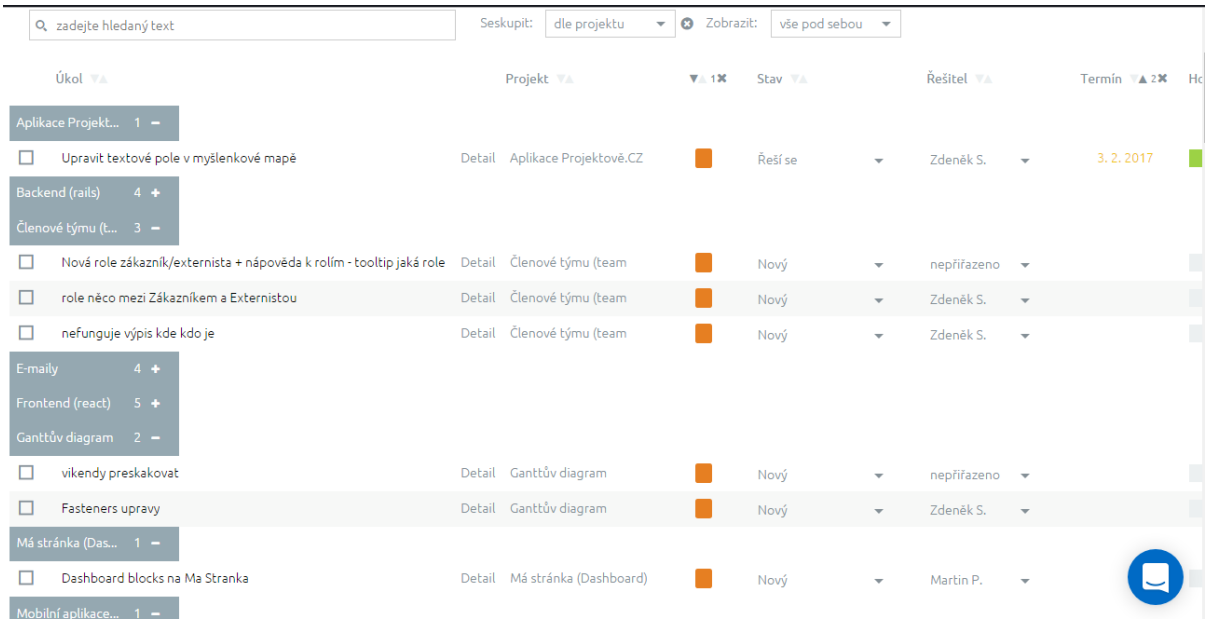
```
export const Grouped = {
  byAssignee(props) {
    let { tasks, users } = props

    return tasks
    .groupBy(t => t.assignedToId && (users.get(t.assignedToId.toString()) ? t.assignedToId : 0))
        .sortBy((_v, k) => {
          let sample = k === 0 ? I18n.t('issues.index.grouped.sample') : null
              , user = k && k !== 0 ? users.get(k.toString()) : null

              return user ? user.fmtShort : sample
              }, (a, b) => {
                if (!a) return 1
                if (!b) return 0
                return comparator(a, b)
              })
  },
```

Výpis 1: Ukázka kódu seskupení dle řešitele

Výsledek, který si vrátím, jsou seskupené úkoly. Tyto úkoly projíždím pomocí funkce *'map'*, kde využívám dva parametry hodnotu a klíč. Potřebuji si z nich vytáhnout název, abych mohl pojmenovat danou skupinu. Opět jsem zde potřeboval udělat vytahování názvu dynamické, aby nezáleželo na tom, jaké seskupení máme. Zde jsem využil *Lodash* funkci *'includes'*, která mi vrátí *true* pokud je daná hodnota v daném poli. Tato funkce byla nejlepší, jelikož seskupení dle autora a řešitele se řeší podobně, seskupení dle dat se řeší podobně a zbytek seskupení se řeší také podobně. U zbytku (pokud má název) stačí jej vrátit. Jinak vracím prázdný řetězec. U dat musím využít knihovny *moment* a funkce *'format'*, abych si vrátil daný formát data dle jazyka uživatele nebo název ze slovníku, když nebylo datum přiřazeno. Ve firmě rozlišujeme pouze anglický a klasický evropský formát dat. U autorů a řešitelů řeším, zda je klíč vetší nebo roven jedné, pak dle zadaného klíče, který je zároveň i id uživatele, si uživatele najdu a vrátím jeho jméno. Jinak pokud je roven nule vrátím z knihovny název pro admina a jinak vrátím ze slovníku, že úkol nebyl přiřazen. V poslední fázi této funkce si zavolám funkci, kterou si vytvořím z úkolů již jednotlivé řádky tabulky s danými sloupci. Nakonec ještě zavolám pomocnou komponentu ***ExpendableGroup***.

Komponenta ***ExpandableGroup*** je sdílená, mezi mnoha komponentami. Proto musí být napsána tak, aby se i později snadno používala. K tomu se využívá *propTypes*, který se používá pro definování typů *props*, které má dostat a zároveň se dá definovat, zda je daný *props* povinný. Dále se dá definovat *defaultProps*, jak už z názvu vyplývá, jde o implicitní hodnoty. Tedy o hodnoty, které se nastaví, pokud nedojde nic v *props*. Funkčnost této komponenty slouží k vytvoření vizualizace skupin. Řešení je následovné ve funkci *'renderHeader'* si definuji hlavičku skupiny. Hlavička má danou šířku, název a číslo z *props*, které zde definuje počet záznamů v dané skupině. Dále je zde ikona plus nebo mínus. Tato ikona je klikací, a zajišťuji s ní to, že můžeme skupinu uzavřít (po kliknutí na ikonu mínus), anebo opětovně otevřít (po kliknutí na ikonu plus). Uzavírání řeším pomocí *state expand*, který je *true* pokud je skupina rozevřená nebo *false* pokud je uzavřená. V *props* si posílám její původní nastavení. Pro moje skupiny je původní nastavení vždy *true*. Další funkce je *'renderContent'*, kterou pokud je skupina otevřená si vrátím data zaslána z *props*, kterými jsou jednotlivé řádky tabulky. Ve funkci *'render*, jelikož je komponenta sdílená, mám více možností. Pokud mám definovaný počet sloupců z *props columns*, tak použiji element **tbody**, kde první řádek má jeden sloupec, kterým je hlavička, kterou jsem si vytvořil ve funkci *'renderHeader'*. Další řádky jsem si vytvořil pomocí funkce *'renderContent'*, kde to tedy jsou jednotlivé řádky úkolů. Tuto možnost využívám právě pro seskupení. Druhou možností je vrátit hlavičku i obsah pouze obalenou v elementu **div**.

Po vytvoření ***ExpandableGroup*** ze seskupených úkolů, se vytvořený virtuální DOM (tj. stromová struktura, která nese informaci o tom, jak má reálný DOM vypadat), vrátí z této funkce a zapíše se do funkce *'render'* v komponentě s výpisem úkolů. Výsledkem je tedy seskupený výpis úkolů, viz Obrázek 10.



Obrázek 10: Výpis seskupených úkolů

5.2.2 Filtrování úkolů

U filtrování bylo hned ze začátku jasné, že bude muset být rozděleno na více komponent. Na začátek jsem si kód rozdělil do čtyř komponent. Poté jsem ještě přidal komponentu pro našeptávání filtrů.

Komponenta ***Filter*** je hlavní komponentou, kterou si vše vykresluji. Starám se v ní o komunikaci s výpisem úkolů, kde mám funkci pro změnu úkolů ve *Store* – tedy kde se volá *Action*. Tato funkce je v komponentě pro výpis úkolů, jelikož zobrazené úkoly ovlivňuje více faktorů, než jenom filtr a seskupení, proto než aby každá komponenta komunikovala zvlášť se *Store*, tak mají společnou funkci v komponentě, která je jejich předkem.

Další komponentou je ***FilterDropdownMenu***, kterou si vykresluji a starám se o ovládání vyskakujícího rámečku po kliknutí do filtru, ve kterém si může uživatel nastavit své filtry nebo vybrat typ filtru.

Třetí komponentou je komponenta ***Select***, kterou si ovládám a vykresluji po rozkliknutí typu filtru jeho další možnosti (například jméno daného řešitele, pro typ filtru řešitel). Komponentou vykresluji pouze již nenastavené filtry. Komponenta mi slouží opět převážně k vykreslování. Jedinou funkčnost, kterou v ní mám je zobrazení dialogu a kalendářů pro nastavení rozmezí pro filtry s daty (termín, změněno a podobně) a volání funkce, kterou si posílám v *props*, pro nastavení daného filtru.

Poslední komponentou je ***Tag***, kterou si vykresluji ve filtru jednotlivé nastavené filtry. Dávám ji tedy data od komponenty ***Filter***, abych zjistil, které filtry jsou nastavené, a vytvořím si pro ně jednotlivé štítky. Většinu funkcí, které v této komponentě využívám, posílám pomocí *props*, komponenty ***Filter*** (například pro smazání filtru).

Jak již z textu vyplývá nejsložitější komponentou je komponenta ***Filter***. Starám se v ní o všechny data, téměř všechny funkce, a spojuji zde dané komponenty do jedné. Nejprve si nastavuji konstanty pro verzi *Local Storage* a pro výchozí nastavení *states*, kde si i nastavuji prázdný filtr. Poté si v konstruktoru zjistím pomocí funkce, zda je filtr obsažen v URL. Tato funkce byla původně pouze v této komponentě, ale nakonec jsem ji přesunul do knihovny funkcí, aby ji mohli použít i kolegové ve firmě pro svoji práci. Pokud se v URL filtr nachází, nastavím jej do filtru pomocí funkce *'getFiltersFromURL'*. V této funkci využívám další svoji funkci, kterou jsem dal do knihovny funkcí a to *'getHash'*, kterou si vrátím daný *hash* dle parametru, zde je parametr **f**. Vytvořil jsem si vlastní regulární výraz, který mi pomáhá se složitějšími filtry – hlavně s daty, které mají složitější zápis a byla potřeba je správně rozparsovat. Poslední, co bylo potřeba vytvořit, než začnu jednotlivé části *hash* projíždět, byla struktura, která mi mapuje filtry. Například pokud se najde v URL písmeno **a** pomocí této struktury si název upravím na stejný, jako je v *state* tedy na **assignedToId** (řešitel). Potom už můžu použít *Lodash* funkci *'forEach'*, kterou si projíždím jednotlivé záznamy pro daný *hash*. Pro pár filtrů zjišťuji zvlášť, zda jsou to ony, jelikož jsou velmi specifické oproti ostatním. Dále aplikuji regulární výraz, a pokud se nalezne shoda pro jeden z filtrů pro data, přidám filtr s datem. Zde již využiji svoje

mapování. Pokud není ani datem, ani specifickým filtrem je jasné, že filtr, který použiji, bude využívat pole a tedy do něj přidám danou hodnotu. Nakonec svůj vytvořený filtr vrátím a ten si uložím do *state*, a nastavím jej.

Pokud nebyl filtr nalezen v URL, nastavím implicitní a zjišťuji si, zda se nachází v *Local Storage*. Pokud ano, zjišťuji, zda se nacházím v projektu, pak nastavím proměnnou *cachedFilters* pomocí *Local Storage* s pojmenováním *filter-ip* (in projects), jinak ji nastavím pomocí *filter-mt* (my tasks). Filtr v *Local Storage* je struktura, která je stejná jako *state*, a díky tomu nemusím nic upravovat. Pokud se nacházel v *Local Storage*, tedy je něco uloženo v proměnné *cachedFilters* nastavím ji do *state*, pomocí *Lodash* funkce *'merge'*. Poté ještě musím nastavit daný filtr do URL.

Pro přidávání do URL opět využívám svého mapování, tentokrát naopak z **assignedToId** si udělám písmeno **a**. A hlavně zde využívám *Lodash* funkce *'reduce*. Tato funkce dokáže procházet kolekci, nad ní udělat určitou operaci a vrátit ji do nové proměnné. Takhle si vytvořím svůj vlastní *hash*, a funkcí *'setHash'* si jej nastavím do URL.

Dále jsem si vytvořil funkce pro manipulaci s filtry, konkrétně pro přidání jednoho, smazání jednoho a smazání všech.

Funkci *'addFilter'* volám poté, co si uživatel zvolí filtr (funkce pro přidání), a v parametrech posílám název typu filtru a jeho hodnotu. Jelikož *state* se nesmí měnit přímo, jinak může nastat, že v danou chvíli bude *state* jiný než očekáváme, tak si *Lodash* funkcí *'cloneDeep'* nakopíruji celou strukturu *state*. Filtr pro data může být vždy jen jeden (například pro termín jeden, pro začátek jeden atd.), takže pokud je název filtru jedním z filtrů pro data využívám *Lodash* funkce *'reject'*, abych si z obecného filtru vytáhl daný filtr. A poté vložím nový filtr. Pokud je datum z rozmezí, ukládám si místo id strukturu s daty. U některých filtrů také přepisuji jejich hodnoty a u ostatních (u většiny), které mají id ne jedno, ale pole id, si dodám přidávané id do pole, jak u obecného filtru, tak u konkrétního. Nakonec této funkce si nastavím pomocí funkce *'setState'* *state*, nastavím filtr do *Local Storage*, do URL a zavolám funkci pro změnu filtru ve *Store*, které v parametru pošlu funkci *'generateCriteria'*. V této funkci musím zjišťovat plno možností, které mohou nastat, viz Výpis 2. Například pokud jsem v otevřených úkolech, nesmí jít nastavit filtr pro stav. Další, co si zde musím nastavit je, že pro typy filtrů, které mohou mít více hodnot, musím zaslat funkci, která tyto hodnoty prochází a to pouze ty, které ještě nejsou přidány. Poté je to podobné jako všude jinde. Jelikož nastává trochu jiná situace pro filtry s daty, pro pár filtrů specifických a pro ostatní. Vygeneruji si zde nakonec kritéria, která potřebuji poslat do funkce pro změnu filtru ve *Store*, ve správném tvaru a pouze za splněných podmínek.

```
1  generateCriteria(queryCriteria=this.props.queryCriteria, query=this.props.query) {
2    let criteria = {}
3      , projectsId
4
5    if (window.projectId) {
6      let project = this.props.projects.find(p => p.id == window.projectId)
```

```
7       projectsId = this.props.projects.filter(p => p.lft >= project.lft && p.rgt <= project.rgt)
            .map(p => p.id)
8     } else {
9       projectsId = this.props.projects.map(p => p.id)
10    }
11    _.each(this.state.filters, f => {
12      if (!_.has(criteria, f.type) && (!_.has(queryCriteria, f.type) || f.type === "projectId"))
            {
13        if (f.type !== 'statusId' || queryCriteria.meta.status === 'open' || typeof queryCriteria
              .meta.status === "function") {
14          if (f.type !== 'projectId' || projectsId.includes(f.value)) {
15            if (this.state[f.type] instanceof Array) {
16              criteria[f.type] = t => _.includes(this.state[f.type], t[f.type])
17            } else {
18                if (f.type == 'dueDate' || f.type == 'createdAt' || f.type == 'startDate' || f.
                      type == 'lastActivityAt' || f.type == 'closedAt') {
19                  if (f.type == 'closedAt' && query != 4) return
20                  let startDate = null
21                    , endDate = null
22                  switch(f.value) {
23                    case 0:
24                      startDate = new Date(moment().startOf('day').valueOf())
25                      endDate = new Date(moment().endOf('day').valueOf())
26                      break;
27                    case 1:
28                      startDate = new Date(moment().startOf('week').valueOf())
29                      endDate = new Date(moment().endOf('week').valueOf())
30                      break
31                    case 2:
32                      startDate = new Date(moment().startOf('month').valueOf())
33                      endDate = new Date(moment().endOf('month').valueOf())
34                      break
35                    default:
36                      startDate = new Date(f.value.start.valueOf())
37                      endDate = new Date(f.value.end.valueOf())
38                      break
39                  }
40                  criteria[f.type] = t => t[f.type] !== null && (new Date(t[f.type]) >= startDate
                        && new Date(t[f.type]) <= endDate)
41                }
42                else if (f.type === 'repetitive' && f.value) {
43                criteria.repetitive = t => t.repetitive !== null
44              } else if (f.type === 'fromFavoriteProjects' && f.value) {
45                _.has(criteria, 'meta') ? criteria.meta.fromFavoriteProjects = true : criteria.
                      meta = { fromFavoriteProjects: true }
46              } else criteria[f.type] = this.state[f.type]
47            }
48          }
```

```
49        }
50      }
51    })
52    if (this.state.subject) {
53      criteria.subject = this.state.subject
54    }
55    return criteria
56  }
```

Výpis 2: Ukázka kódu generování kritérií dle nastavených filtrů pro *Store*

Funkci *'deleteFilter'* volám poté, co uživatel klikne na křížek u daného štítku filtru. V parametru si opět posílám název typu filtru a jeho hodnotu. Opět si zde pomocí *Lodash* funkce *'cloneDeep'* nakopíruji *state*. U filtrů s daty pouze smažu filtr. U specifických filtrů je nastavím na výchozí hodnotu. A u filtrů, které jsem vytvořil jako pole hodnot, opět využívám *Lodash* funkci *'reject'*, ale tentokrát hodnotu neměním, ale nastavím ji na prázdnou, takže ji smažu. Nakonec nastavím *state*, nastavím filtr do *Store* pomocí funkce a změním URL a *Local Storage*.

Poslední funkcí pro změnu filtrů je *'deleteAll'*, která se volá, když uživatel klikne na větší křížek u rámečků pro filtry. Ve funkci si opět nakopíruji *state*, celý ho nastavím na výchozí hodnotu, nastavím jej také do *Store* a smažu z URL a z *Local Storage*.

Další funkcí v této komponentě je funkce pomocí, které hledám v názvech úkolů. Pokud uživatel napíše tři písmena do vstupu nebo zmáčkne enter, nastaví se filtr a vyhledám dané úkoly.

Poslední důležitá funkce je *'render'*. V této funkci si vykresluji celou komponentu ***Filter***. Nastyloval jsem ji tak, aby pokaždé když uživatel klikne do rámečku s filtry, se objevil rámeček těsně za posledním přidaným filtrem. Ikony jsem styloval absolutně vůči rámečku filtru celému.

Jak jsem již zmínil v komponentě pro výpis úkolů je funkce, ve které volám *Action* pro změnu zobrazených úkolů ve *Store*. V této komponentě se zároveň udržuje *state* s názvem *filter*, který obsahuje nejen můj filtr, ale i jiná nastavení pro zobrazení úkolů. Ve funkci zjistím, zda se změnil *Lodash* funkcí *'isEqual'*. Pokud ano, zavolám funkci v *Action* s novým filtrem. Tato funkce se pouze propadne do *Store*, kde je již tato funkce definována. Funkce, kterou jsem zavolal, změní ve *Store* filtr na nový. Ale jelikož se filtr aktualizoval, detekuje se změna, a zavolá se tak funkce *'getVisibleTasks'*, která filtruje všechny úkoly, které jsou v *state* tohoto *Store*, a nastaví se nové viditelné úkoly ve *state* tohoto *Store*, a všechny komponenty, která na tyto úkoly momentálně naslouchají, tyto úkoly dostanou, a vykreslí je.

Další komponentou je komponenta ***FilterDropdownMenu***, zde si vyfiltruji a seřadím uživatele a projekty pro komponentu ***Select***. Z uživatelů filtruji pouze neaktivní uživatele. U projektů řeším, zda se v nějakém nacházím. Pokud ano, můžu filtrovat jen podle něj a jeho potomků, proto si vyfiltruji jen tyto projekty, a poté je seřadím. Pokud nejsem v projektu, projekty pouze seřadím.

Dále v této komponentě řeším, který název si pošlu do komponenty ***Select***, abych věděl, která data vykreslit (řešitele, projekty a podobně). Ve funkci *'isActive'* řeším, zda můžu zobrazit daný

typ filtru. Jelikož například v Mé úkoly nemůžu měnit řešitele, a podobně, a proto tyto typy vůbec nezobrazím. Některé typy nemají další možnosti, a proto nepoužívají komponentu ***Select***. Tak je rovnou pomocí funkce *'addFilter'*, kterou jsem si poslal v *props*, přidám do filtru.

Ve funkci *'render'* mám obrovský *switch*, který umožňuje provést jen určitou část kódu, dle nějakého parametru, zde je parametrem typ filtru. V *switch*, tedy po kliknutí na daný typ filtru, měním data pro ***Select***. Smažu z dat pro ***Select***, ty která jsou již nastavená ve filtru, pomocí negované *Lodash* funkce *'includes'*, která mi vrátí *true*, pokud je již daná hodnota obsažena ve filtru. Případně smažu i typy filtrů, které by neměly být vidět v tomto rámečku. Tedy filtry, které se nepoužívají v komponentě ***Select***, tentokrát s *Lodash* funkcí *'findIndex'*, která vrací mínus jedna, pokud se daná hodnota již nachází v obecném filtru ve *state*.

Ve funkci *'render'* si vykreslím vše, co se dá přidat a navěsím na ty, na které potřebuji komponentu ***Select***.

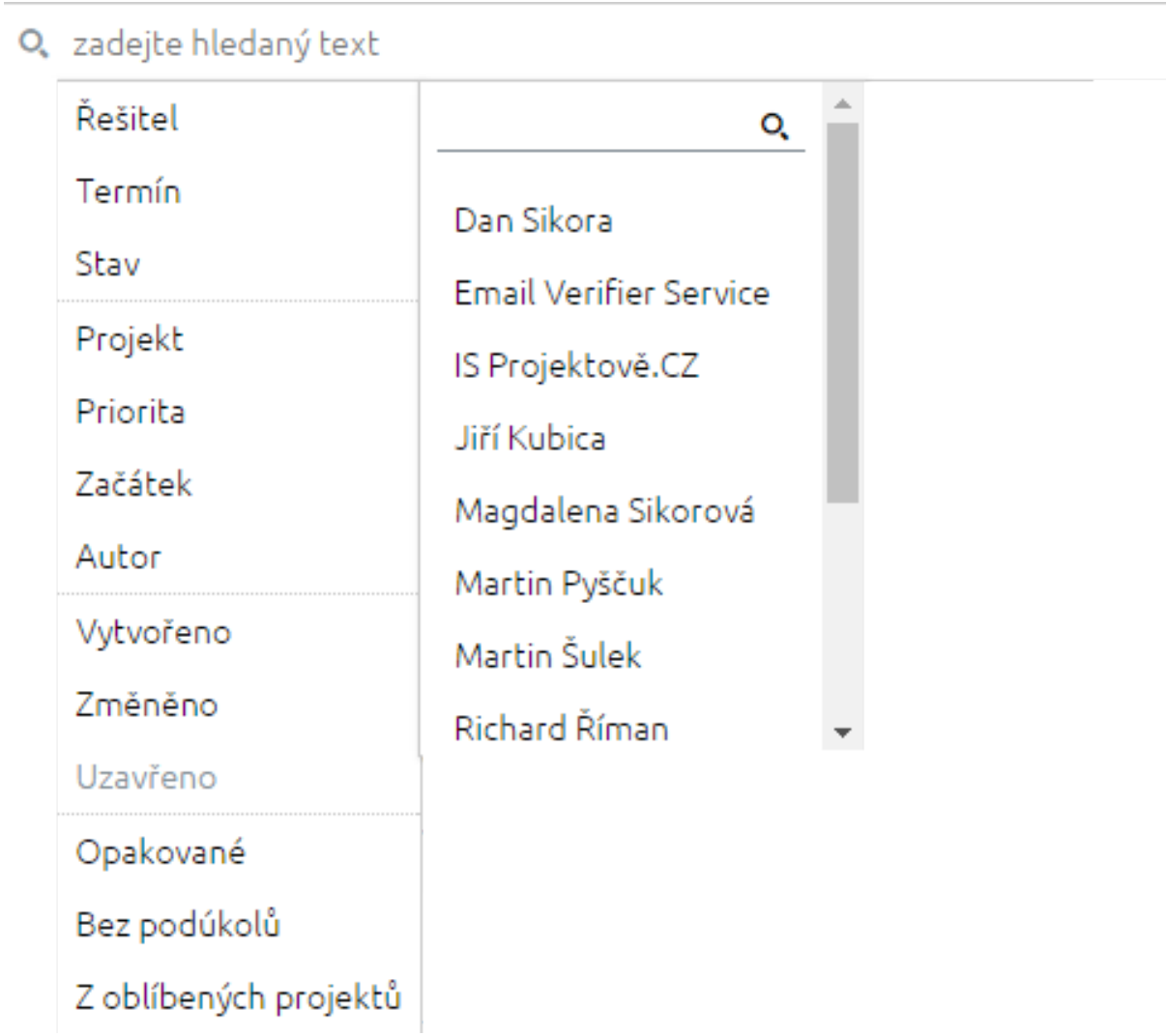
V komponentě ***Select*** neřeším příliš funkčnosti, ta hlavní je z *props* pro přidání filtru, ale hlavně, zde vykresluji dle *props* z komponenty ***FilterDropdownMenu***, co může uživatel ještě přidat.

Komponenta má pouze tři funkce. Jednou z nich je funkce *'update'*, která vrací upravená data, která vykresluji do tohoto rámečku pomocí funkce *'searchMembers'*.

Ve funkci *'searchMembers'* vyhledávám v uživatelích a projektech. Využívám ji tedy pouze pro ***Select*** řešitel a projekt. Využívám zde funkce *'indexOf'*, která pokud mi vrátí mínus jedna, tak daný řetězec neodpovídá hledanému. Proto ji vyfiltruji z dat, která zobrazuji. Poslední věc v této funkci, kterou řeším je, když se nenajde žádný uživatel nebo projekt, nastavím si v *state* této komponenty *noSearch* na *true*, a zobrazím v tomto případě uživateli, že nebyl nalezen žádný uživatel nebo projekt s tímto jménem.

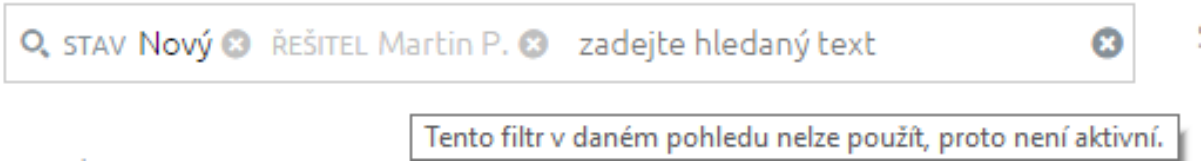
Poslední funkcí je funkce, kterou ovládám zobrazení dialogu, ve kterém je komponenta ***DateRangePicker***, kterou si zobrazím dva kalendáře mezi, kterými jde vybírat rozmezí dat.

Na konec tedy komponenty ***FilterDropdownMenu*** a ***Select*** zobrazují typy filtrů a jejich hodnoty (pokud nějaké mají), viz Obrázek 11.



Obrázek 11: Ukázka výsledku komponent ***FilterDropdownMenu*** a ***Select*** pro výběr filtru

Předposlední komponentou je komponenta ***Tag***, které jsem dal všechny funkce do funkce *'render'*, jelikož slouží pouze pro vykreslení filtrů, a nebyla potřeba vytvářet více funkcí. Je zde opět funkce *'isActive'*, jelikož pokud si uživatel v Otevřených úkolech nastaví filtr s řešitelem, a přejde na Mé úkoly. Zde tento filtr nemůže být aktivní, změní se mu tedy styly, aby uživatel ihned věděl, proč má nastavený jiný filtr pro řešitele, než si sám nastavil, viz Obrázek 12.



Obrázek 12: Ukázka aktivního a neaktivního filtru

Poslední funkcí je *'getValue'*. Tato funkce je tu, jelikož ve filtru mám nastavené id, a nikoliv

jméno uživatele, projektu a podobně. Proto, zde ve *switch* řeším dle jména filtru, kde mám hledat, a najdu si z obecných dat (všichni uživatelé, všechny projekty a podobně) dle shody s id, co hledám, a vrátím si jeho název nebo jméno.

Ve funkci *'render'* už jenom procházím nastavené filtry, které si posílám v *props*, a vykresluji je pomocí daných funkcí.

Poslední komponentou, kterou požívám u filtru je ***FilterAutocomplete***. Tuto komponentu jsem dodával později, jelikož je to něco navíc, a původně se s ní nepočítalo. Jak již název napovídá, komponenta slouží pro vyhledávání filtrů, aby nemusel uživatel naklikávat, ale například jen napsat jméno uživatele, a vybrat si jej buď jako řešitele, nebo jako autora, nebo napsat i typ filtru a vybrat si jej rovnou. V této komponentě mám čtyři funkce včetně funkce *'render'*. V komponentě využívám *React* funkci *'componentWillReceiveProps'* pro komponenty, ve které si kontroluji, zda mi nedošla nová nebo jiná data v *props*, abych je měl stále aktuální. U této komponenty přijímám slova ze vstupního pole z komponenty ***Filter***. Proto jsem musel zajistit, abych vyvolal rámeček s našeptávačem až po třech písmenech, kdy už si nezobrazuji komponentu ***FilterDropdownMenu***.

První z funkcí komponenty ***FilterAutocomplete*** je *'createIndex'*, kde si na začátku vytvářím konstanty, a jednou z nich je opět struktura pro mapování. Ve funkci dále používám *Lodash* funkci *'each'* pro procházení pole, které mám definované jako konstantu s názvy typů, které vypadají podobně. Do *indexu*, který je *Immutable Set*, si přidávám *Immutable Map*, která má název typu filtru, id, název nebo jméno a nakonec *fulltext*, který využívám pro vyhledávání. Pro *fulltext* používám knihovnu, pomocí které si ze slov vymažu veškerou diakritiku, a pomocí funkce *'toLowerCase'* zmenším všechna písmena. Dále projíždím všechny uživatele, pomocí obyčejné, a ne *Lodash* funkce *'forEach'*. Protože jsou typu *Immutable*, a opět si je přidávám do *indexu* jako *Immutable Map*, a se stejnou strukturou. Stejně tak procházím konstantu, kde mám názvy typů filtrů pro data, jelikož se přidávají stejně. Poslední konstanta, kterou procházím je *OTHERS*, kde mám ostatní typy filtrů. V této *'forEach'* funkci si kontroluji, zda není typ *parentId*, jelikož pro něj je id jiné, než pro ostatní. Používám na to ternární operátor, místo klasické podmínky. Tuto funkci si volám při načítání *state* s parametrem všech *props*, a pomocí této funkce si tedy nastavuji *state* s názvem *index*.

Další důležitá funkce je *'search'*. Nejdříve si všechna slova, která mi dojdou, vyfiltruji tak, aby měli minimální délku tři písmena, a uložím si je pomocí funkce *'split'* jako pole slov do proměnné. Do další proměnné si uložím *state index*. Pokud je velikost pole slov rovna jedné, vyhledávám slovo jak v hodnotách filtrů, tak v názvech typu filtrů.

Vyhledávání probíhá pokaždé stejně. S *Lodash* funkcí *'each'* procházím pole slov, a v něm filtruji jednotlivé výsledky, pomocí funkce *'filter'*. Na začátku je výsledkem celý *state index*, který se postupně filtruje. Ve funkci *'filter'* si vytvářím, pomocí vlastní funkce regulární výraz, který mi změní všechny speciální znaky na znaky, se kterými umí funkce pracovat. Hledané slovo si opět pozměním pomocí knihovny pro smazání diakritiky, a zmenším si slovo na malá písmena pomocí regulárního výrazu. Hotový regulární výraz porovnávám na shodu s jednotlivými položkami v

state index, a vracím jen ty, které měli shodu. Po nalezení všech připojím k výsledku vyhledávání v typech. Je téměř stejné jenom si neprojíždím *index*, ale projíždím pole, kde mám jména typů filtrů psaná anglicky v poli díky, kterým se poté dostanu k jménům typů ve slovníku, takže hledání funguje, dle nastaveného jazyka uživatele. Opět zde využívám regulární výraz, knihovnu pro smazání diakritiky, funkci *'toLowerCase'* a *Lodash* funkci *'each'*. Opět si vracím z funkce pouze shody s regulárním výrazem.

Pokud je slov více, hledám pouze první slovo v názvu typu filtrů. Pokud není žádná shoda, výsledek je prázdný, do výsledku si opět uložím celý *index*, a všechna slova postupně vyhledávám v názvech filtrů nikoliv již v typech. Hledání je opět stejné. Jediný rozdíl je v tom, že hledám nejdříve první slovo, a z tohoto hledání mám nějaký výsledek. Až na tomto výsledku znovu hledám druhé slovo atd. Pokud by po vyhledávání v typech shoda byla, vymažu si první slovo z pole hledaných slov, a zbytek slov vyhledám na již omezené množině výsledků, kde je pouze daný typ filtru.

Nakonec funkce ještě vymažu všechny výsledky, které již nelze přidat, jelikož jsou již přidané, a seskupím si vyhledané výrazy pomocí funkce *'groupBy'* podle typu filtrů.

Velmi jednoduchou funkcí je funkce *'add'*, ve které si volám funkci pro přidání filtru poslanou v *props*, po kliknutí uživatelem na vyhledaný výraz.

Funkce *'render'* je velmi podobná funkci *'render'* v komponentě ***FilterDropdownMenu***, je jen malinko upravená stylově. Výpis probíhá následovně. Pomocí funkce *'map'* procházím všechny typy, abych si vypsal názvy typů, které nejsou prázdné. V předchozí funkci *'map'* ty, které nejsou prázdné, procházím další funkcí *'map'*, abych si vypsal nalezené výsledky. Konečný výsledek můžeme vidět na Obrázku 13.

Obrázek 13: Ukázka našeptávače pro filtr

Všechny tyto komponenty pro filtrování proběhli mnoha úpravami kódu ať už kvůli opakujícím se částem nebo proto, že daná část šla napsat lépe.

6 Popis vedlejších úkolů a jejich řešení

6.1 Hlavní menu a React Router

Pomocí tohoto úkolu jsem se učil první dny *React*. Komponenty jsou to velmi jednoduché, ale pro začátečníka ideální na pochopení *React*. Dále jsem u těchto komponent poprvé bral data ze *Store*, ze serveru a poprvé jsem více styloval. Tento úkol mě naučil tedy základy *React* a prohloubil mé znalosti CSS.

Poslední fází bylo připravit komponentu ***React.Router***, která umožňuje rychlejší a plynulejší přechod mezi jednotlivými stránkami. Po nastudování jsem komponentu také nachystal, ale bohužel tuto komponentu nebylo možné ještě nasadit, protože jednotlivé stránky musí být všechny napsané v *React*, aby fungovala.

6.2 Dodání sloupců do výpisu úkolů a položek do detailu úkolu

Do firmy jsem přišel v době, kdy začal přechod na *React*, a s tím přišel nový výpis úkolů a nový detail úkolu. Detail úkolu se již neotevírá na nové stránce ale pouze jako dialog. Zde se projevilo výborné využití komponent, kdy se pro výpis i pro detail pro danou položku použila stejná komponenta.

Podobnými komponentami jsou zde autor, projekt, řešitel a nad-úkol. U těchto komponent jsem použil komponentu ***react-select***, která poskytuje propracovanější *select*. Mým úkolem bylo tedy ji využít, poskytnout jí data a vyřešit změny v *select*. Takže poskytnout změnu do *Store* a na server. Všechna data musela být před vložením vyfiltrována, každá trochu jinak, a to dle práva daného uživatele. U komponent autor a řešitel jsem jako první ve firmě využil nemít vybranou možnost a možnosti, které jsou na výběr jako pouhý text, ale využil jsem komponenty ***Avatar***, která poskytuje obrázek nebo iniciály uživatele a dále k němu přidal i jméno. K tomu jsem využil funkcí *'customOption'* a *'cutomValue'* z komponenty ***react-select***.

Následující komponenty se využívají pouze u detailu úkolu. První z nich jsou sledující. Komponenta obsahuje vždy řešitele daného úkolu, který jako jediný nejde smazat. Dále se dají přidat ostatní uživatelé pomocí dialogu. Dialog má výpis uživatelů, kteří mají právo vidět tento úkol. Dá se mezi nimi vyhledávat, zaškrtnout všechny a další funkce. Uživatelé se ze sledujících dají odebrat buď pomocí dialogu, nebo kliknutím na ně. Sledující uživatelé dostávají informace o změnách v úkolu.

Dále jsem přidal komponentu pod-úkoly a související úkoly, která v detailu vypisuje tyto úkoly a umožňuje je odebrat nebo přidat další.

Poslední komponentou jsou náklady. Tato komponenta jako jediná nekomunikuje se *Store*, takže se všechna data musí brát ze serveru. Ale u této komponenty to nevadí, protože se nepoužívá tak často. Opět je zde dialog pro vytvoření nových nákladů, umožňující dát k nim i poznámku a datum.

6.3 Nahrávání souborů k úkolu a galerie

Tento úkol zahrnoval i výpis souborů, ale vzhledem k tomu, že jsem již podobný výpis dělal u pod-úkolů a souvisejících úkolů, nebylo to pro mě nic složitého.

Druhou částí tohoto úkolu bylo nahrávání a mazání souborů. Nahrávat úkol jde nyní pomocí tlačítka, kdy vám vyjede dialog, kde si vyberete soubory, a dáte uložit nebo pomocí přetažení souborů do prohlížeče. V části s dialogem jsem musel vyřešit, jak nahrát více souborů zároveň. Pro nahrávání pomocí přetažení souborů do prohlížeče, tedy do tzv. *dropzone*, jsem použil komponentu ***react-dropzone***, ale byla potřeba hodně úprav od stylování. Původně byla *dropzone* malá, ale byla potřeba ji mít přes celou obrazovku, a zároveň dát uživateli najevo, že je *dropzone* aktivní, a může vložit soubory. Druhá úprava byla v odchytávání, zda uživatel drží soubory nad naší stránkou. V komponentě ***react-dropzone*** tato část nefungovala příliš dobře, a byla potřeba, vyřešit to tak, že jsem musel sám odchytávat události *dragEnter*, *dragLeave* a *drop*. Další problém se vyskytl později, a bylo jím, že *dropzone* odchytávala i soubory nebo cokoliv, co uživatel chytl na obrazovce. Vyřešil jsem to pomocí parametru události *dataTransfer.types*, kdy jsem odchytával pouze typy *Files*.

Poslední částí tohoto úkolu byla galerie pro soubory, které jsou obrázky. První část, kterou jsem musel vyřešit, bylo načtení pouze obrázků. Vyřešil jsem to podle typu přípon u souborů. Poté jsem si udělal vlastní náhled na obrázky. Pokud uživatel klikl na daný obrázek, otevřela se mu galerie na daném obrázku. Pro galerii jsem použil komponentu ***react-gallery***. Tuto komponentu jsem použil v dialogu a komponentě jsem předal, na kterém obrázku má začít, dále všechny obrázky ve struktuře se jménem a podobně. Galerii jsem musel opět malinko stylově upravit, aby seděla v dialogu správně.

6.4 Připomínání úkolů s časem

Jelikož je služba Projektově.CZ velmi často využívána na hlídání termínu u úkolů, bylo potřeba nějak připomínat úkoly s časem. Řešení bylo takové, že jakmile se blížil konec nějakého úkolu, zobrazil se, ještě nad hlavičkou stránky, rámeček s časem. V rámečku bylo, kdy má být úkol dokončen a název úkolu. Tento rámeček šel buď zavřít celý nebo z něj jen nějaký úkol odebrat.

Musel jsem si tedy vzít všechny úkoly ze *Store*, a vyfiltrovat je. Nejdříve dle řešitele, aby uživatel viděl jen své úkoly. Poté ještě dle časů. Úkoly byly rozděleny na více časových rozmezí, konkrétně na pár minut před koncem úkolu a na pár minut po konci úkolu. Tyto dvě skupiny měly různý vzhled. Poté už je stačilo jen vypsat. Pokud ale uživatel daný úkol nechtěl mít zobrazený, stačilo jej odebrat. Dané odebrání se uložilo do *Local Storage*, a úkoly, které jsou v *Local Storage*, se musely vyfiltrovat z výpisu. Další problém u tohoto úkolu byl ten, že se musíme starat, aby *Local Storage* nebyla příliš velká, a proto se úkoly, které nebyly v původních vyfiltrovaných úkolech, dle řešitele a času, ale byly v *Local Storage*, musely smazat z *Local Storage*.

7 Využité a scházející znalosti

7.1 Využité teoretické a praktické znalosti získané v průběhu studia

Při práci jsem využil několik znalostí ze studia, ale pokaždé to byli pouze základy.

Určitě mi pomohl předmět Skriptovací a programovací jazyky, kde jsme probírali framework Django, který stejně jako aplikace Projektově.CZ, využívá modelu MVC.

Dalšími předměty byly Programování I a II, Algoritmy I a II a Programovací jazyky I, odkud jsem využil znalosti objektového programování, znalost řadících a vyhledávacích algoritmů, a znalost struktur jakými jsou List, Set a podobně.

V neposlední řadě pro mě byly nejdůležitějšími předměty Tvorba aplikací pro mobilní zařízení I a Vývoj internetových aplikací. V těchto předmětech jsem získal znalosti jak teoretické, tak praktické o fungování webových aplikací, a hlavně praktické znalosti s používáním HTML, CSS a JavaScriptu.

Tyto znalosti mi velmi pomohly hlavně ze začátku odborné praxe.

7.2 Scházející teoretické a praktické znalosti v průběhu odborné praxe

Nejvíce chybějící znalostí pro mě byla znalost frameworku React. React má mnoho možností, a nebylo pro mě vůbec jednoduché, pobrat tolik novinek.

Další chybějící znalostí pro mě bylo využívání cizích knihoven, jakými jsou Lodash, Immutable, ale i například využití cizí komponenty, která se v praxi často používá. Díky tomu jsem se naučil číst anglické dokumentace a návody.

Chyběly mi také znalosti preprocesorů CSS a CSS metodologie BEM.

Poslední ale také velmi důležitou znalostí, která mi scházela, byla znalost verzovacího systému, konkrétně jsem na odborné praxi využíval Git. Verzovací systém se využívá, pokud více lidí pracuje na stejné aplikaci, a pokud je potřeba oddělit verze hotového kódu a kódu ve vývoji. Před odbornou praxí jsem neměl ani ponětí, že něco takového existuje.

8 Zhodnocení a dosažené výsledky

Odborné praxe, kterou jsem měl možnost absolvovat, si velice cením, a hodnotím ji jako jednu z nejlepších zkušeností v době studia.

Firma mi poskytla všechny materiály ke studiu, a při učení na prvních úkolech mi poskytla výbornou podporu, když jsem si nevěděl rady. Při učení na prvních úkolech se také průběžně dívali na můj kód. Ukázali a vysvětlili mi, jak některé věci napsat efektivněji nebo přehledněji.

Poprvé jsem se zde setkal s mnoha novinkami, ať už jím byl framework React nebo verzovací systém Git, které jsou moderní, a velmi často používané ve všech firmách zabývajícími se webovými aplikacemi. Odborná praxe mi také dala možnost podílet se poprvé na tak rozsáhlé aplikaci, a poprvé pracovat v programátorském týmu, což mě naučilo psát kód přehledněji a pečlivěji tak, aby kódu porozuměli i kolegové.

Práce, na které jsem se podílel, je velmi silným nástrojem pro reporting úkolů. Tyto reporty klienti velmi často poptávali, a i když jej služba Projektově.CZ obsahovala, ve starém řešení bylo jejich použití příliš neohrabané. S novým, nutno říct rychlejším, a uživatelsky příjemnějším řešením jsou klienti, dle jejich názorů, velmi spokojeni.

Literatura

[1] Interní dokumentace firmy

[2] React - virtuální DOM a navázání funkcí (bind)
Dostupné z: http://reactkungfu.com/

[3] React dokumentace
Dostupné z: https://facebook.github.io/react/docs/

[4] Store a Action - zdroj č. 1
Dostupné z: http://alt.js.org/docs/

[5] Store a Action - zdroj č. 2
Dostupné z: http://redux.js.org/docs/api/

[6] Immutable
Dostupné z: https://facebook.github.io/immutable-js/

[7] Novinky JavaScript ES6
Dostupné z: https://webapplog.com/es6/

[8] Lodash
Dostupné z: https://lodash.com/